Fine-grain Task Aggregation and Coordination on GPUs

Marc S. Orr[†§] Bradford M. Beckmann[§] Steven K. Reinhardt[§] David A. Wood[†§]

[†]University of Wisconsin–Madison
Computer Sciences
{morr,david}@cs.wisc.edu

[§]AMD Research
{brad.beckmann,steve.reinhardt}@amd.com

Abstract

In general-purpose graphics processing unit (GPGPU) computing, data is processed by concurrent threads executing the same function. This model, dubbed single-instruction/multiple-thread (SIMT), requires programmers to coordinate the synchronous execution of similar operations across thousands of data elements. To alleviate this programmer burden, Gaster and Howes outlined the channel abstraction, which facilitates dynamically aggregating asynchronously produced fine-grain work into coarser-grain tasks. However, no practical implementation has been proposed.

To this end, we propose and evaluate the first channel implementation. To demonstrate the utility of channels, we present a case study that maps the fine-grain, recursive task spawning in the Cilk programming language to channels by representing it as a flow graph. To support data-parallel recursion in bounded memory, we propose a hardware mechanism that allows wavefronts to yield their execution resources. Through channels and wavefront yield, we implement four Cilk benchmarks. We show that Cilk can scale with the GPU architecture, achieving speedups of as much as 4.3x on eight compute units.

1. Introduction

Graphics processing units (GPUs) are gaining popularity for general-purpose, high-performance computing because GPU architectures tolerate recent technology trends better than CPU architectures. The GPU's data-parallel design, which amortizes front-end hardware across many threads, is more area- and power-efficient than the massive caches and complex speculation logic that typify CPUs. By dedicating transistors to as many simple threads as possible, GPUs are suited better to continue capitalizing on Moore's Law [1].

Many leading manufacturers now integrate CPUs and GPUs on the same die, producing what AMD calls accelerated processing units (APUs) [2][3][4]. This coupling is paving a path for improved architectural integration. For example, Heterogeneous System Architecture (HSA) incorporates a unified virtual address space and coherent shared memory spanning the APU, the ability of GPUs to spawn tasks, and user-level task queues for low offload latencies [5]. These features enable new general-purpose GPU (GPGPU) applications and are finding support in languages like CUDA 6 [6] and OpenCL 2.0 [7].

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

Despite this progress, GPUs continue to be confined to structured parallelism, which requires programmers to coordinate independent threads capable of executing the same function at the same time. Structured parallelism maps directly to the GPU's data-parallel hardware, but many unstructured applications cannot take advantage of the GPU.

Channels, outlined by Gaster and Howes [8], are multi-producer/multi-consumer data queues that have potential to expand GPGPU programming. Channels reside in virtual memory and act as a medium through which producers and consumers communicate in a data-flow manner. A given channel holds fine-grain data items—which we call channel elements (CEs)—that are processed by the same function. Constraining each channel to be processed by exactly one function facilitates efficient aggregation of work that then can be scheduled onto the GPU's data-parallel hardware.

While Gaster and Howes defined channels, they did not propose an implementation, leaving designers to question their practicality. To this end, we propose and evaluate the first implementation of channels. We find that GPU threads often write the same channel at the same time. Thus, we begin by developing a channel data structure that is lock-free, non-blocking, and optimized for single-instruction-multiple-thread (SIMT) accesses.

The finer-grain parallelism enabled by channels requires more frequent and complex scheduling decisions. To manage this behavior, we leverage the existing task-scheduling hardware in today's GPUs, which typically is implemented as a small, in-order, programmable processor, rather than fixed-function logic. We use this tightly integrated processor to monitor the channels, manage algorithmic dependencies among them, and dispatch ready work to the GPU. Our analysis suggests that replacing the existing in-order processor with a modest out-of-order processor can mitigate the scheduling overheads imposed by dynamic aggregation.

Because no existing programs are written specifically for channels, we evaluate our implementation by mapping flow graph-based programs to channels. A flow graph is a data-driven graph representation of a parallel application. It is a popular abstraction used by many modern parallel programming languages, including Intel's Threading Building Blocks (TBB) [9]. Flow-graph nodes represent the program's computation, while messages flowing over directed edges represent communication and coordination. We use channels to aggregate individual messages into coarser-

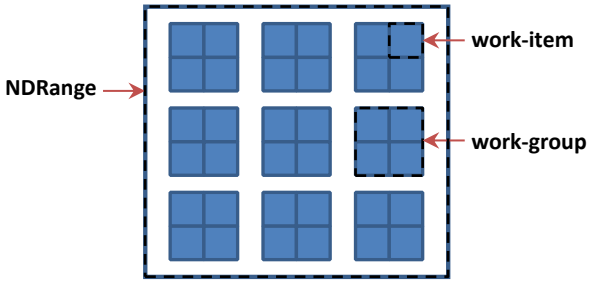

Figure 1: OpenCL NDRange

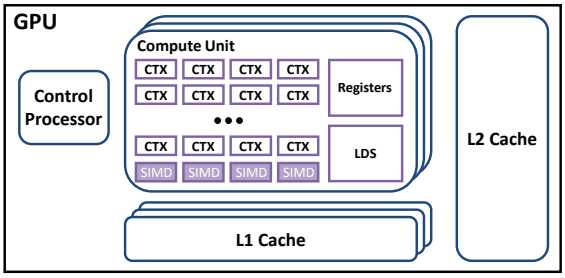

Figure 2: Generic GPU architecture

grain units that can be scheduled efficiently onto the GPU. Channel-flow graphs increase the diversity of applications that map well to GPUs by enabling higher-level programming languages with less rigid task abstractions than today's GPGPU languages.

We specifically explore mapping programs written in Cilk to channels. Cilk is a parallel extension to C/C++ for expressing recursive parallelism. We define a set of transformations to map a subset of Cilk to a channel-flow graph so that it can execute on a GPU. This presented two important challenges. First, GPUs do not provide a call stack, which CPUs normally use to handle recursion. Our solution is to map Cilk's task tree to "stacks of channels". Second, previous Cilk runtimes use depth-first recursion to bound memory usage. However, although breadth-first scheduling is more effective at populating a GPU's thousands of hardware thread contexts, it requires exponential memory resources [10]. To solve this problem, we propose a bounded breadth-first traversal, relying on a novel yield mechanism that allows wavefronts to release their execution resources. Through channels and wavefront yield, we implement four Cilk workloads and use them to demonstrate the scalability of Cilk in our simulated prototype.

To summarize, our contributions are:

- We enable efficient fine-grain task scheduling on GPUs by proposing the first channel implementation and associated hardware support.
- We propose a mechanism for GPU wavefronts to yield their execution resources, enabling wavefronts to spawn tasks recursively while bounding memory consumption.
- Using channels and wavefront yield, we enable Cilk on GPUs and show that its performance scales with the architecture.

2. GPU Architecture and Programming Model

This section gives an overview of today's GPU programming abstractions and how they help programmers coordinate structured parallelism so that it executes efficiently on the GPU's data-parallel hardware.

2.1 GPU Programming Model

The GPU's underlying execution resource is the single-instruction/multiple-data (SIMD) unit, which is a number of functional units, or lanes, that execute in lockstep (64 on AMD GPUs and 32 on NVIDIA GPUs [6]). GPGPU languages, like OpenCL™ and CUDA, are called SIMT because they map the programmer's view of a thread to a SIMD lane. Threads executing on the same SIMD unit in lockstep are called a wavefront (warp in CUDA). In SIMT languages, a task is defined by three components:

1. A function (called a kernel).
2. Data (the kernel's parameters).
3. A dense 1- to 3-dimensional index space of threads called an NDRange (grid in CUDA).

Figure 1 shows an OpenCL NDRange. The smallest unit is a work-item (thread in CUDA), which is an SIMT thread that maps to a SIMD lane. Work-items are grouped into 1- to 3-dimensional arrays called work-groups (thread blocks in CUDA). Multiple work-groups are combined to form the NDRange. The NDRange helps programmers schedule structured parallelism to the GPU's data-parallel hardware, but makes mapping unstructured parallelism difficult.

2.2 GPU Architecture

Figure 2 highlights important architectural features of a generic GPU. Compute units (CUs, called streaming multiprocessors in CUDA), are defined by a set of SIMD units, a pool of wavefront contexts (CTX), a register file, and a programmer-managed cache called local data store (LDS, or shared memory in CUDA). A CTX maintains state for an executing wavefront. Each wavefront owns a slice of the register file that partially defines its state. Each CU has a private L1 cache that feeds into a shared L2 cache. While today's GPUs and APUs provide some combination of coherent and incoherent caches, next-generation GPUs and APUs that adhere to HSA will provide a fully coherent cache hierarchy [5].

The control processor, also shown in the generic GPU architecture picture (Figure 2), obtains SIMT tasks from a set of task queues that it manages. To schedule a task, the control processor assigns its work-groups to available CUs. The control processor also coordinates simultaneous graphics and compute, virtualizes GPU resources, and performs power management. To carry out its many roles, this front-end hardware has evolved from fixed function logic into a set of scalar processors managed by firmware.

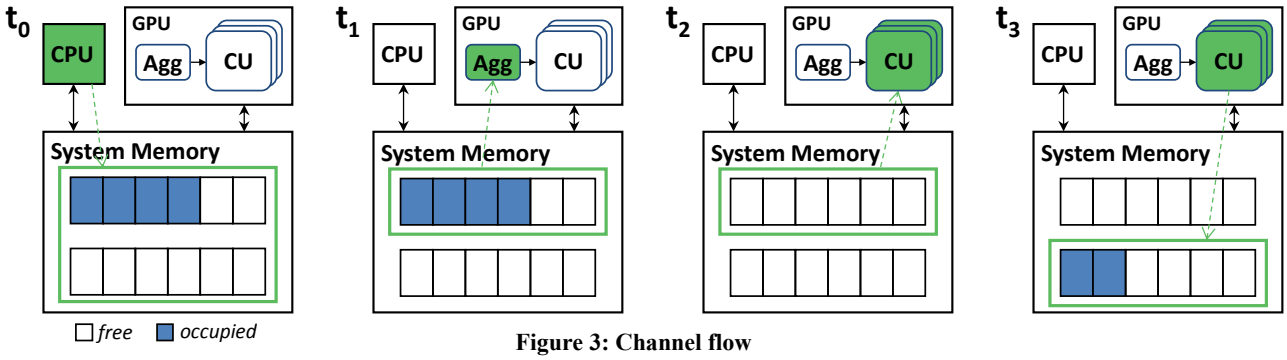

Figure 3: Channel flow

3. Channel Definition and Implementation

Gaster and Howes suggested channels to improve on today's coarse-grain GPU task abstractions. In this section, we summarize their vision and propose the first channel implementation, which executes on forward-looking APUs.

3.1 Prior Work on Channels

A channel is a finite queue in virtual memory, through which fine-grain data (channel elements, or CEs) are produced and consumed in a data-flow manner. Channels resemble conventional task queues, but differ in three ways:

1. Data in a channel is processed by exactly one function permanently associated with that channel.
2. CEs are aggregated dynamically into structured, coarse-grain tasks that execute efficiently on GPUs.
3. Each channel has a "predicate" function for making dynamic scheduling decisions.

Figure 3 shows data moving through channels in an APU-like system that includes a CPU and a GPU connected to a coherent shared memory. The GPU's control processor is extended to monitor and manage channels. Because we are concerned primarily with this new capability, we call the control processor the aggregator (labeled *Agg* in Figure 3) for the remainder of this paper.

At time 0 (t_0) in Figure 3, the host initializes two channels and populates them with CEs. At time 1 (t_1), the aggregator, controlled through a user-defined scheduler, probes the channels; it detects enough CEs to justify a dispatch to GPU hardware. The GPU consumes the CEs at t_2 and produces new CEs in a different channel at t_3.

Restricting each channel to processing by exactly one function avoids burdening the aggregator with inspecting individual CEs. This constraint does not limit fine-grain task-parallelism because channels are mapped to shared virtual memory and therefore are visible to all producers.

The predicate is a Boolean function that assists the aggregator in making scheduling decisions. The simplest predicate is one that returns false unless enough CEs are available to populate all of a SIMD unit's lanes. This is what we assume for this paper.

3.2 Lock-free Channel Implementation

To realize finer-grain task abstractions on GPUs, we introduce a novel multi-producer/multi-consumer queue that is lock-free, non-blocking, and array-based. Lock-free queues have a rich history in the context of CPUs. Early work considered array-based designs [11][12][13], but linked lists are preferred [14][15]. Linked lists are not well suited for GPUs because different work-items in a wavefront consuming adjacent CEs are susceptible to memory divergence, which occurs when the work-items access different cache blocks; if the requests had been to the same cache block, the GPU's coalescing hardware could have merged them. We find that our queue implementation accommodates the high levels of contention that are typical on a massively threaded GPU.

3.2.1 Array-based Channel Implementation

Our array-based channel is implemented as three structures:

1. **Data array:** Buffer for produced CEs.
2. **Control array:** Buffer of data-array offsets, populated by producers and monitored by the aggregator.
3. **Done-count array:** Adjacent data-array elements can share a done-count element. The aggregator monitors the done-count array to free data-array elements in the order they were allocated.

The size of the done-count array is the size of the data array divided by the number of data-array elements that share a done count. The control array is twice the size of the data array. Array elements can be in one of five states:

1. **Available:** Vacant and available for reservation.
2. **Reserved:** Producer is filling, hidden from aggregator.
3. **Ready:** Visible to aggregator, set for consumption.
4. **Dispatched:** Consumer is processing.
5. **Done:** Waiting to be deallocated by aggregator.

Figure 4 illustrates two wavefronts, each four work-items wide, operating on a single channel in system memory. For space, the control array is the same size as the data array in the figure, but in practice it is twice the size of the data array. In the text that follows, producers operate on the tail end of an array and consumers operate on the head end.

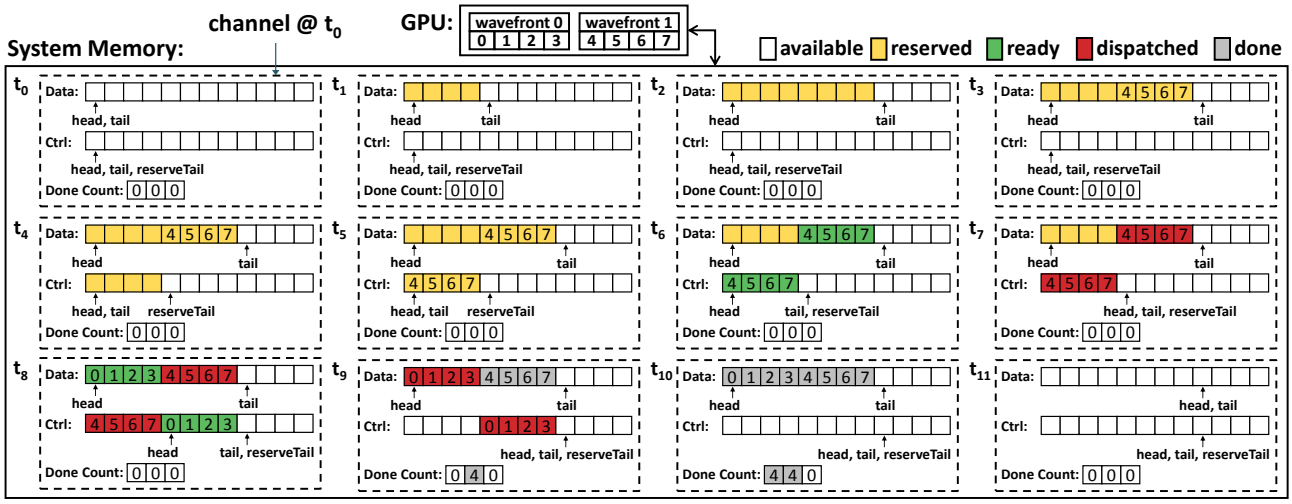

Figure 4: Lock free, non-blocking, array-based channel implementation

At time 0 (t_0), the data array's `head` and `tail` pointers are initialized to the same element. Similarly, the control array's `head` and `tail` pointers are initialized to the same element. The control array maintains two tail pointers (`tail` and `reserveTail`) because producers cannot instantaneously reserve space in the control array and write the data-array offset. All done counts are initialized to 0.

At t_1, each work-item in wavefront 0 reserves space for a CE. Four data-array elements are transitioned to the reserved state by updating the data array's `tail` pointer via compare-and-swap (`CAS`). At t_2, each work-item in wavefront 1 reserves space for a CE and at t_3 those work-items finish writing their data-array elements.

Data-array elements are made visible to the aggregator by writing their offsets into the control array. Specifically, at t_4, wavefront 1 updates `reserveTail` via `CAS` to reserve space in the control array for its data-array offsets. At t_5, the offsets are written and at t_6 the control array's `tail`, which is monitored by the aggregator, is updated to match `reserveTail`. The array elements related to wavefront 1 are now in the ready state. The design is non-blocking because wavefront 1 can make its CEs visible to the aggregator before wavefront 0 even though it reserved space after wavefront 0.

At t_7, the data-array elements generated by wavefront 1 are transitioned to the dispatched state when the aggregator points consumers at their respective control-array elements. Those control-array elements also transition to the dispatched state; they cannot be overwritten until their corresponding data-array elements are deallocated because the control array is twice the size of the data array.

At t_8, wavefront 0 finishes writing its data-array elements and makes its CEs visible to the aggregator. At t_9, wavefront 0's CEs are dispatched. Also at t_9, the consumers of wavefront 1's CEs signal that they no longer need to reference the data-array elements by updating their respective done counts atomically; these data-array elements cannot be deallocated before wavefront 0's data-array elements. At t_{10}, the consumers of wavefront 0's CEs update their respective done counts. Finally, at t_{11}, the aggregator deallocates space.

```
 1: int gpuReserveNElements(int numEl, int *tail) {
 2:   int wfTail = 0;
 3:   // 1. Choose one work-item to operate on tail
 4:   bool update = most_sig_work_item();
 5:   // 2. Intra-wavefront prefix sum
 6:   int offset = prefix_sum(numEl);
 7:   int numElToRes = offset + numEl;
 8:   // 3. Intra-wavefront synchronization
 9:   join_wfbarrier();
10:   while(update) {
11:     int oldTail = *tail;
12:     int nextTail = oldTail + numElToRes;
13:     int curTail = CAS(tail, oldTail, nextTail);
14:     if(oldTail == curTail) {
15:       wfTail = oldTail;
16:       update = false;
17:     }
18:   }
19:   wait_at_wfbarrier();
20:   // 4. Broadcast tail to entire wavefront
21:   wfTail = reduction(wfTail);
22:   return (wfTail + offset);
23: }
```

Figure 5: GPU fetch-and-update (ignores wrapping/overflow)

3.2.2 *Discussion and Optimization*

The array-based channel maps well to the GPU's coalescing hardware. The CUs are responsible for allocation and consumption while the aggregator handles deallocation, which is off the critical path of execution. The aggregator manages the channels without inspecting their individual CEs.

Space is reserved in the data and control arrays through conditional fetch-and-update (via `CAS`). By leveraging intra-wavefront communication instructions [16][17], this operation can be amortized across a wavefront, greatly reducing memory traffic. Figure 5 depicts pseudo-code with these optimizations that updates a channel array's tail pointer.

```
 1: #define LEN 32768
 2:
 3: typedef struct {
 4:   int val;
 5: } FibObj;
 6:
 7: void FibKernel(int srcID, int srcOff,
 8:                int destID, int *result) {
 9:   FibObj *src = (FibObj *)deq(srcID, srcOff);
10:   if(src->val <= 2) {
11:     atomic_add(result, 1);
12:   } else {
13:     FibObj *ob = (FibObj *)talloc(destID, 2);
14:     ob[0].val = src->val - 1;
15:     ob[1].val = src->val - 2;
16:     enq(destID, ob);
17:   }
18:   tfree(srcID, srcOff);
19: }
```

```
20: void main(int argc, char * argv[]) {
21:   int n = atoi(argv[1]);
22:   int res = 0;
23:
24:   Graph g;
25:   ChannelNode *ch = g.ChannelNode(sizeof(FibObj), LEN);
26:   KernelNode *kern = g.KernelNode(FibKernel);
27:   kern->setConstArg(2, sizeof(int), ch->chID);
28:   kern->setConstArg(3, sizeof(int *), &res);
29:   ch->connectToKernelNode(kern);
30:
31:   FibObj *ob = (FibObj *)ch->talloc(1);
32:   ob->val = n;
33:   ch->enq(ob);
34:
35:   g.execute();
36:   g.waitForDone();
37:   printf("fib(%d) = %d\n", n, res);
38: }
```

Figure 6: Fibonacci example

4. Programming with Channels

This section proposes a low-level API to interface channels and describes utilizing channels through flow graphs.

4.1 Channel API

Table 1 shows the channel API. Producers call `talloc` to allocate CEs. An allocated CE is made visible to the aggregator via the `enq` function. A CE must be enqueued to the channel that was specified during its allocation. A consumer obtains work with the `deq` function; the specific channel and offset within that channel are managed by the aggregator. After data is consumed, the aggregator is signaled that deallocation can occur via the `tfree` API.

The `talloc` API enables minimum data movement between producers and consumers because the destination channel is written directly through the pointer that `talloc` returns. Figure 6, lines 3-19, demonstrate the API in Table 1.

4.2 Channel-flow Graphs

Flow graphs comprise a set of nodes that produce and consume messages through directed edges; the flow of messages is managed through conditions. Several popular parallel programming languages and runtimes support flow graphs. For example, Intel's TBB provides a sophisticated flow-graph abstraction [9]. MapReduce and StreamIt provide more constrained flow-graph frameworks [18][19]. GRAMPS, which had a strong influence on the original proposal for channels, explores scheduling flow graphs onto graphics pipelines [20].

Channels facilitate flow graphs with fine-grain messages. A channel-flow graph is specified as a directed graph composed of kernel nodes and channel nodes. A kernel node resembles a GPGPU kernel that consumes and produces data. Kernel nodes are analogous to function nodes in TBB. A channel node is a buffer that accumulates messages produced by kernel nodes and routes them to be consumed by other kernel nodes. Channel nodes are similar to queue nodes in TBB, but bounded.

Table 1: Channel API

API Function	Description
`void *talloc(int id,int cnt)`	allocate `cnt` CEs in channel `id`.
`void enq(int id,void *ptr)`	place CEs at `ptr` in channel `id`.
`void *deq(int id,int off)`	get CE in channel `id` at `off`.
`void tfree(int id,int off)`	free CE in channel `id` at `off`.

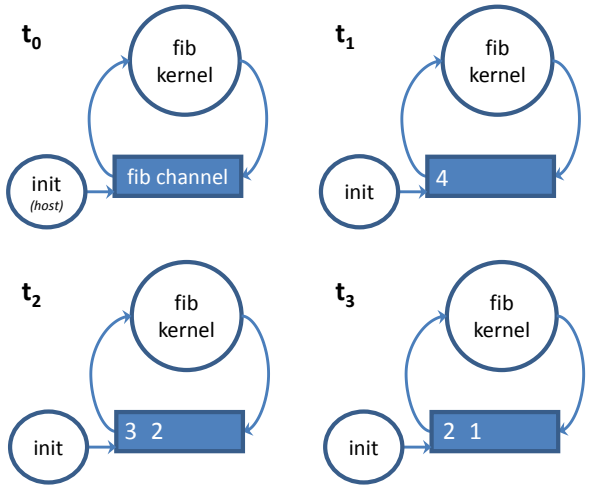

Figure 7: Channel-flow graph for naïve Fibonacci

Figure 7 shows an example flow graph to compute the fourth Fibonacci number. At time 0 (t_0), the graph, made of one kernel node and one channel node, is initialized on the host; the CE is uniquely defined for that channel. At t_1, an `init` node (the host) puts source CEs in the channel node. At t_2, the kernel node consumes CEs from the channel node and produces new CEs. At t_3, the kernel node consumes the remaining CEs and the computation is done.

A simple graph API was prototyped for this research. Figure 6 demonstrates how to build the channel-flow graph shown in Figure 7. A sophisticated flow-graph framework is beyond the scope of this work. The remainder of this paper focuses on other aspects of our design.

5. Case Study: Mapping Cilk to Channels

Channels facilitate mapping higher-level abstractions to GPUs. As an example, we discuss translating a subset of the Cilk programming language to a channel representation.

5.1 Cilk Background

Cilk extends C/C++ for divide-and-conquer parallelism [21]. Cilk programs use the keyword `spawn` before a function to schedule it as a task. The keyword `sync` forces its caller to block until all of its spawned tasks are complete. Figure 8 demonstrates how these keywords are used to calculate the *n*th Fibonacci number. These two Cilk primitives form the basis of the language and are what we explore mapping to channels. Other primitives are left for future work.

```
1: int fib(int n) {
2:   if(n <= 2) return n;
3:   else {
4:     int x = spawn fib(n - 1);
5:     int y = spawn fib(n - 2);
6:     sync;
7:     return (x + y);
8:   }
9: }
```

Figure 8: Fibonacci in Cilk

5.2 Cilk as a Channel-flow Graph

One strategy to implement Cilk on channels is to divide kernels into sub-kernels that are scheduled respecting dependencies. Specifically, a sub-kernel is created whenever `sync` is encountered. Each `spawn` is translated into a `talloc/enq` sequence that reserves space in the correct channel, writes the task parameters, and schedules the work. Each `sync` is translated into a `talloc/enq` sequence that schedules work to a channel connected to the "post-sync" sub-kernel. It may be possible to automate these translations, but they are done manually for this research.

Figure 9 shows the Cilk tree to calculate the fifth Fibonacci number. Shaded circles are "pre-spawn" tasks (lines 2-5 in Figure 8). White circles are "post-spawn" tasks (line 7 in Figure 8). Solid lines depict task spawns and dotted lines are dependencies. Each circle is labeled with a letter specifying the order in which it can be scheduled.

Shaded circles, or pre-spawn tasks, have no dependencies. They are labeled "A" and are scheduled first. White circles, or post-spawn tasks, depend on shaded circles and other white circles. Dependencies on shaded circles are respected by scheduling white circles after all shaded circles are complete. White circles are labeled "B" or lexicographically larger. Dependencies among white circles are inferred conservatively from the level of recursion from which they derive. For example, the white circle representing the continuation for the fourth Fibonacci number and labeled "C" derives from the second level of the Cilk tree and depends on a continuation that derives from the third level.

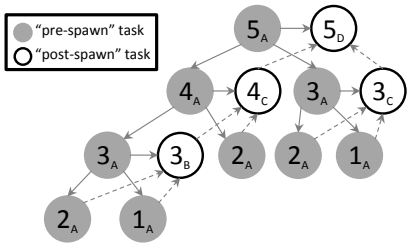

Figure 9: Cilk tree for Fibonacci

Continuations that derive from deeper levels of the Cilk tree can be scheduled first. This is achieved by maintaining "stacks of channels" for continuations and scheduling each continuation at the correct offset within the stack. Virtual memory is allocated up front for channel stacks, similar to how CPU threads are allocated private stacks. Tasks determine the correct offset within the stack by accepting their recursion depth as a parameter. The scheduler drains the channel at the top of the stack before scheduling channels below it. This strategy is called levelization [22].

Figure 10 shows the tasks from Figure 9 organized into a main channel for pre-spawn tasks and a stack of channels for post-spawn tasks.

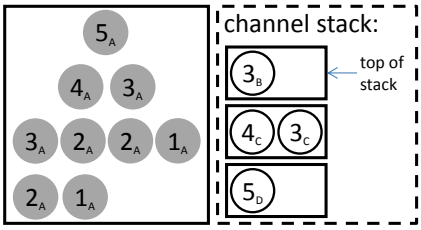

Figure 10: Managing dependencies with channels

Figure 11 shows the channel-flow graph for the Cilk version of Fibonacci. Channel stack nodes (e.g., the dashed box in Figure 11) are added to the channel-flow-graph framework. Instead of atomically updating a global result, as is done by the flow graph in Figure 7, each thread updates a private result in the channel stack. Intermediate results are merged into a final result by a second continuation kernel node.

Finally, it should be noted that the translations described for the Cilk version of Fibonacci generalize to other Cilk programs because they all have one logical recursion tree.

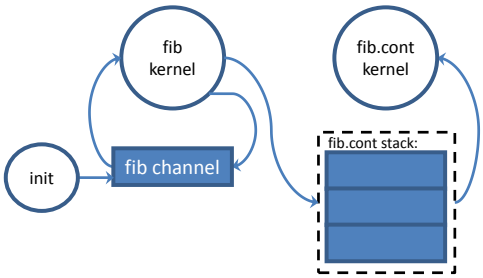

Figure 11: Channel-flow graph for Cilk version of Fibonacci

5.3 Bounding Cilk's Memory Footprint

For CPUs, Cilk runtimes use a work-first scheduling policy to bound the memory footprint to the depth of the Cilk tree. In work-first scheduling, threads traverse the Cilk tree in a depth-first manner by scheduling the continuation for a task that calls `spawn` and executing the spawned task [21]. This does not generate work fast enough for GPUs.

The scheduling policy described in Section 5.2 is called help-first. It generates work quickly by doing a breadth-first traversal of the Cilk tree, but consumes exponential memory relative to a workload's input size [10]. To make this policy feasible, the memory footprint must be bounded. This is possible if hardware supports yielding a CTX.

If a hardware context yields its execution resources when it is unable to obtain space in a channel, the scheduler can drain the channels by prioritizing work deeper in the recursion. When a base-case task is scheduled, it executes without spawning new tasks, freeing space in its channel. When a task near the base case executes, it spawns work deeper in the recursion. Because base-case tasks are guaranteed to free space, forward progress is guaranteed for the recursion prior to the base case. Inductively, forward progress is guaranteed for all channels.

The scheduler can differentiate work from different recursion levels if both pre- and post-spawn tasks are organized into channel stacks, as shown in Figure 12.

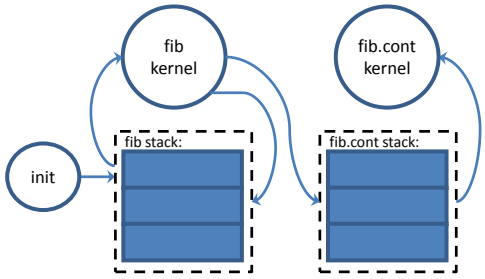

Figure 12: Bounding memory for the Cilk version of Fibonacci

An alternative approach is a hybrid scheduler that uses help-first scheduling to generate work and then switches to work-first scheduling to bound memory [23]. Future work will compare a help-first only scheduler to a hybrid scheduler.

6. Wavefront Yield

To facilitate Cilk and similar recursive models, we propose that future GPUs provide a "wavefront yield" instruction. Our yield implementation, depicted in Figure 13, relies on the aggregator to manage yielded wavefronts. After a wavefront executes yield (❶), the GPU saves all of its state to memory (❷) including registers, program counters, execution masks, and NDRange identifiers. LDS is not saved because it is associated with the work-group and explicitly managed by the programmer; a restarting wavefront must be assigned to the same CU on which it was previously executing. Memory space for yield is allocated for each CTX before dispatch and deallocated as wavefronts complete. This is the same strategy used for spill memory in HSA.

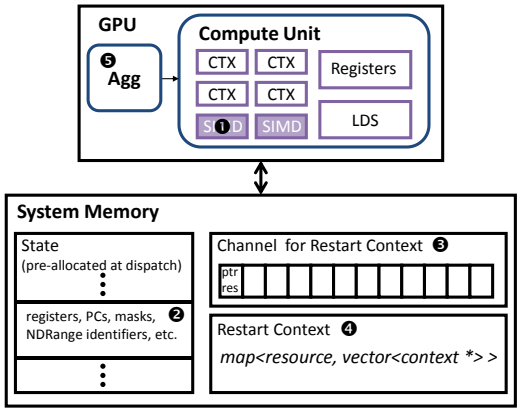

Figure 13: Wavefront yield sequence

In addition to the wavefront's state, a restart context, used to restart the wavefront, is saved to a data structure in memory (❸). This data structure can be a finite size because the aggregator will consume whatever is inserted into it; in our implementation, we use a channel. The restart context comprises a pointer to the wavefront's saved state and the resource that the wavefront blocked on. The aggregator retrieves the restart context and inserts it into a software-defined data structure that tracks blocked wavefronts (❹). The aggregator then schedules a new wavefront to occupy the yielded context (❺). The aggregator monitors resources and restarts wavefronts as appropriate.

7. Methodology and Workloads

We prototyped our channel implementation in the simulated system depicted in Figure 14. We used gem5 [24] enhanced with a proprietary GPU model. The GPU's control processor is implemented as a programmable core that serves as the aggregator. It is enhanced with private L1 caches that feed into the GPU's unified L2 cache. Each CU has a private L1 data cache that also feeds into the GPU's L2 cache. All CUs are serviced by a single L1 instruction cache connected to the GPU's L2 cache. More details can be found in Table 2.

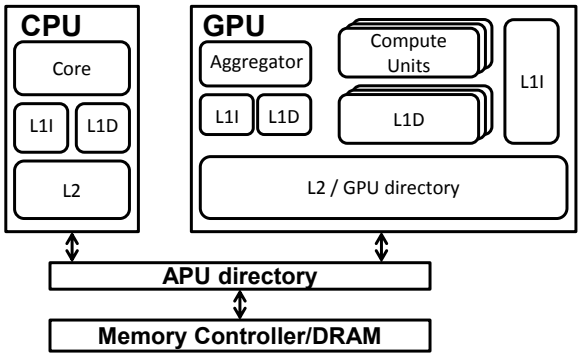

Figure 14: Simulated system

To isolate the features required for channels, all caches are kept coherent through a read-for-ownership MOESI directory protocol [25] similar to the GPU coherence protocol proposed by Hechtman et al. [26]. Future work will evaluate channels with write-combining caches [27].

We implemented wavefront yield as described in Section 6. CTXs require, at a minimum, 856 bytes for program counters, execution masks, NDRange identifiers, etc. Additional bytes are required for registers. There are three kinds of registers: 64 x 4 byte (s), 64 x 8 byte (d), and 64 x 1-bit (c). The number of registers varies across kernels. We save all registers (live and dead). A more sophisticated implementation would avoid saving dead registers. The numbers of registers for our workloads are shown in Table 3. In the worst case (Queens), 9,072 bytes are saved/restored.

7.1 Workloads

We wrote four Cilk workloads derived manually from Cilk source according to the transformations discussed in Section 5. They are characterized in Table 3.

1. **Fibonacci:** Compute the *n*th Fibonacci number. Partial results are stored in continuation channels and merged by a continuation kernel.
2. **Queens:** Count the number of solutions to the NxN queens puzzle. In our implementation, derived from code distributed with the MIT Cilk runtime [21], the base case is a 4x4 sub-section of the chessboard.
3. **Sort:** Recursively split an array into four smaller sub-arrays until reaching a base case (64 elements), sort all of the base-case sub-arrays, and merge them. This workload also was derived from a version distributed with the MIT Cilk runtime [21].
4. **Strassen:** Repeatedly divide a matrix into four sub-matrices down to a base case (16x16 elements), multiply each pair of base-case matrices, and combine the results through atomic addition [28].

7.2 Scheduler

A scheduling algorithm, which executes on the aggregator, was written for the Cilk workloads. It respects Cilk's dependencies by tracking the current level in the recursion, as described in Section 5.2. It also checks for wavefronts that have yielded and restarts them as resources (i.e., channels deeper in the recursion) become available. Because levelization enforces dependencies, the GPU can block on the scheduler. We explore workload sensitivity to the aggregator in Section 8.3.

Table 2: Simulation configuration †See Section 8.3

Compute Unit	
Clock	1GHz, 4 SIMD units
Wavefronts (#/scheduler)	40 (each 64 lanes)/round-robin
Data cache	16kB, 64B line, 16-way, 4 cycles, delivers one line every cycle
Instr. cache (1 for all CUs)	32kB, 64B line, 8-way, 2 cycles
Aggregator	
Clock	2GHz, 2-way out-of-order core†
Data cache	16kB, 64B line, 16-way, 4 cycles
Instr. cache	32kB, 64B line, 8-way, 2 cycles
Memory Hierarchy	
GPU L2/directory	1MB, 64B line, 16-way, 16 cycles
DRAM	1GB, 30ns, 20GB/s
Coherence protocol	MOESI directory
Host CPU (not active in region of interest)	
Clock	1GHz, gem5 TimingSimpleCPU
L1D, L1I, L2 (size/assoc/latency)	64B lines across all caches (64kB/2/2), (32kB/2/2), (2MB/2/2)
Channel	
Done count	64 (Section 8.2.1)

8. Results

We find that three of our four Cilk workloads scale with the GPU architecture, and show details in Figure 15. The average task size (average cycles/wavefront) for each workload, shown in Table 3, and cache behavior, depicted in Figure 16, help explain the trends.

First, we examine the workload that does not scale up to eight CUs: Fibonacci. Given the small amount of work in its kernel nodes, we would not expect Fibonacci to scale. Even so, it is useful for measuring the overheads of the channel APIs because it almost exclusively moves CEs through channels. A consequence of Fibonacci's small task size is that it incurs more GPU stalls waiting on the aggregator than workloads with larger task sizes. Larger task sizes allow the aggregator to make progress while the GPU is doing compute. At eight CUs, Fibonacci's cache-to-cache transfers degrade performance; these occur because consumer threads execute on different CUs than their respective producer.

The other three Cilk workloads scale well from one CU to eight CUs, with speedups ranging from 2.6x for Sort to 4.3x for Strassen. This is because the workloads perform non-trivial amounts of processing on each CE, which is reflected in the average task size. Strassen's wavefronts are approximately 37 times larger than Fibonacci's. In contrast, Sort's

Table 3: Workloads †Section 8.2.2 ††Measured from a one-wavefront execution (channel width=64, input=largest with no yields)

Workload	Data set	Kernel nodes	Registers/kernel	Channel width†	# of wavefronts	Average cycles/wavefront††
Fibonacci	24	2	16s/8d/2c, 3s/4d/1c	32,768	2,192	7,046
Queens	13x13	2	16s/8d/3c, 5s/5d/1c	16,384	1,114	35,407
Sort	1,000,000	4	16s/8d/2c (all 4 kernels)	32,768	4,238	30,673
Strassen	512x512	1	16s/6d/8c	8,192	587	259,299

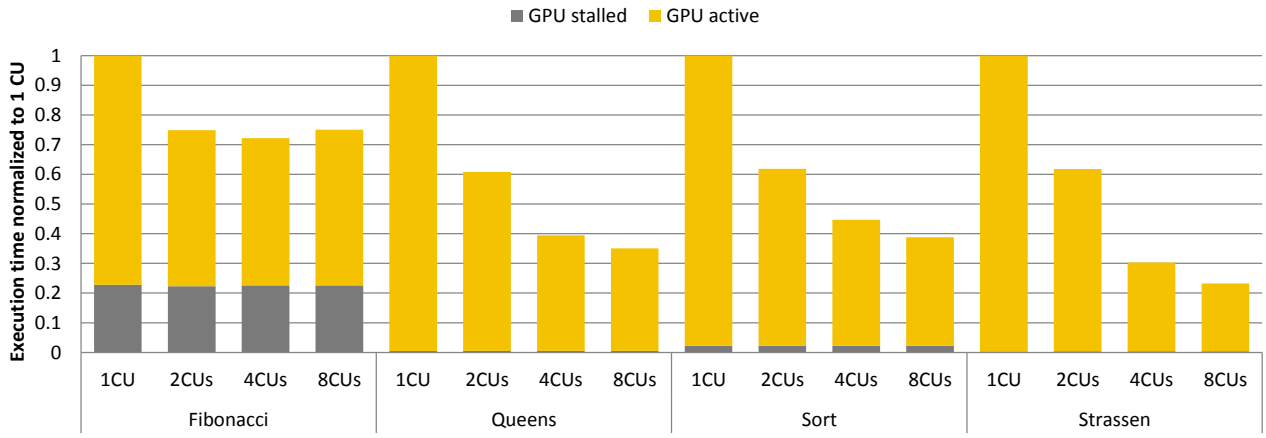

Figure 15: Scalability of Cilk workloads

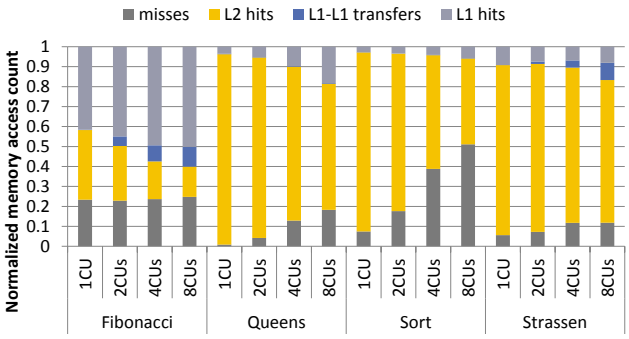

Figure 16: CU cache behavior

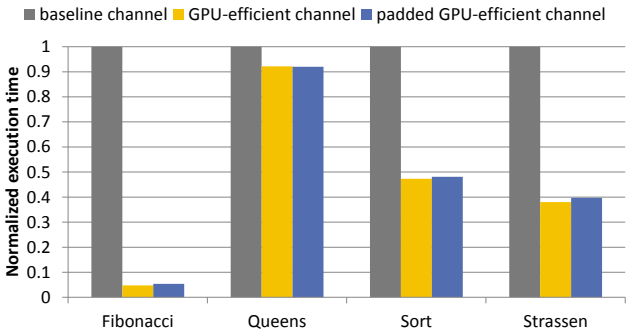

Figure 17: GPU-efficient array quantified

wavefronts are a little more than four times larger than Fibonacci's, indicating that relatively small tasks can be coordinated through channels to take advantage of the GPU.

While few memory accesses hit in the L1 cache, many hit in the shared L2, facilitating efficient communication between producers and consumers. L2 cache misses degrade scalability because main memory bandwidth is much lower than cache bandwidth. As illustrated in Figure 15, the aggregator overhead is constant with respect to the number of CUs, so we would not expect it to be a bottleneck for larger inputs.

To help put these results in context, we compare channel workloads to non-channel workloads when possible. Specifically, we compare Strassen to matrix multiply from the AMD SDK [29] and Queens to a version of the algorithm distributed with GPGPU-Sim [30]. We would expect channels to be slower than conventional GPGPU code because their fine-grain nature leads to more tasks, which imposes extra coordination overhead; both channel codes trigger more than 10 times the number of dispatches than their non-channel counterparts. Surprisingly, we find that channels are on par with conventional GPGPU code because they facilitate more efficient algorithms. Specifically, Strassen has a lower theoretical complexity than AMD SDK's matrix multiply. Meanwhile, for Queens the GPGPU-Sim version pays large overheads to flatten a recursive algorithm that is expressed naturally through channels. Both Strassen and Queens have fewer lines of code (LOC) than the non-channel versions. These results are summarized in Table 4.

8.1 Array-based Design

Figure 17, which compares the baseline channel to a "GPU-efficient channel" that has the intra-wavefront optimizations suggested in Section 3.2.2, shows the effectiveness of amortizing synchronization across the wavefront. By reducing the number of `CAS` operations (where `talloc` and `enq` spend most of their time) by up to 64x, this optimization reduces the run-time drastically for all workloads.

We compared the GPU-efficient channel to a version that is padded such that no two CEs share a cache line. Padding emulates a linked list, which is not likely to organize CEs consumed by adjacent work-items in the same cache line. In all cases, the padded channel performs worse, but the degradation is less than expected because the CEs are organized as an array of structures instead of a structure of arrays. We plan to address this in future work.

Table 4: GPU Cilk vs. conventional GPGPU workloads

	LOC reduction	Dispatch rate	Speedup
Strassen	42%	13x	1.06
Queens	36%	12.5x	0.98

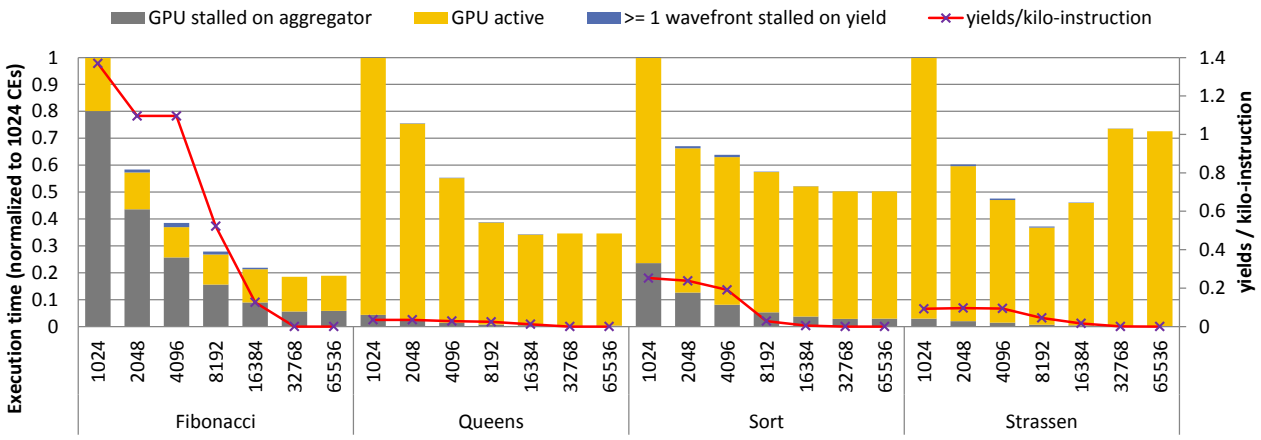

Figure 18: Channel width (CEs)

8.2 Channel Granularity

8.2.1 Done Count

Channel space is deallocated in order at the granularity of the done count. A done count of 64 limits deallocation to less than 3% of total stall time on average.

8.2.2 Channel Width

Figure 18 shows how channel width can affect performance. Narrow channels are unable to supply enough CEs to utilize the GPU adequately. Meanwhile, larger channels degrade the performance of Strassen because wavefronts are not able to use the L2 cache as effectively. We configured each workload with the channel width that resulted in peak performance (shown in Table 3). Better cache-management policies, like cache-conscious thread scheduling [31], may eliminate the cache thrashing caused by wider channels.

8.2.3 Wavefront Yield

Figure 18 also shows the frequency and impact of yields. Saving and restoring CTXs generally has little impact on GPU active time because yields are relatively infrequent. However, at smaller channel widths, frequent yields increase GPU stall time because the aggregator manages yields instead of dispatching new work.

8.3 Aggregator Sensitivity Study

We performed a sensitivity study to determine how complex the aggregator needs to be. The first design that we considered is a primitive core, called *simple*, which is not pipelined and executes one instruction at a time; this extremely slow design increases pressure on the aggregator. We also considered a complex out-of-order (OoO) core, called *4-way OoO*, to capture the other extreme. Finally, we looked at two intermediate designs: *2-way OoO* and *2-way light OoO*. *2-way OoO* resembles a low-power CPU on the market today. *2-way light OoO* is derived by drastically slimming *2-way OoO* and provides insight into how an even simpler core might perform. Table 5 summarizes our findings. *4-way OoO* provides little benefit relative to *2-way OoO*. *2-way light OoO* reduces the performance gap between *simple* and *2-way OoO*, but the aggregator overhead can still be as high as 35%. Hence, *2-way OoO* strikes a good balance between performance and core complexity and was used to generate the results reported in previous sections.

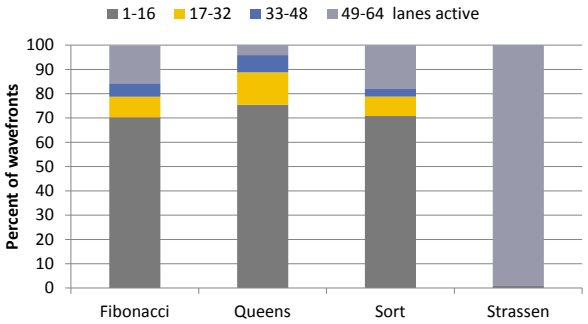

Figure 19: Branch divergence

8.4 Divergence and Channels

Figure 19 depicts branch divergence. Fibonacci and Queens have many wavefronts with base-case and non-base-case threads, leading to high divergence. Strassen has little divergence because it distributes work very evenly. Sort, which spends most of its time in the base case, suffers severe divergence. This is because the base-case code was obtained from a CPU version that uses branches liberally.

Table 5: % of time GPU (8 CUs) is blocked on aggregator

	Description	Fibonacci	Queens	Sort	Strassen
Simple	*no pipelining, one instruction at a time*	41.5	2.9	8.9	3.4
2-way light OoO	*physical registers: 64, IQ size: 2, ROB size: 8, ld/st queue size: 8/8*	35.1	2.0	7.2	2.5
2-way OoO	*physical registers: 64, IQ size: 32, ROB size: 64, ld/st queue size: 32/32*	30.1	1.6	5.8	1.8
4-way OoO	*physical registers: 128, IQ size: 64 ROB size: 128, ld/st queue size: 64/64*	29.8	1.5	5.6	1.9

9. Related Work

We survey three categories of related work: multi-producer/multi-consumer queues, dynamic aggregation of fine-grain work on data-parallel hardware, and GPU task runtimes and synchronization.

9.1 Multi-producer/Multi-consumer Queues

Prior work on lock-free, multi-producer/multi-consumer queues is skewed towards CPUs; it includes linked list- and array-based designs. Linked lists often are preferred because they are not fixed-length and are easier to manage [14][15]. Unfortunately, linked lists are a poor fit for the GPU's memory-coalescing hardware.

Array-based queues often require special atomic operations, limit the size of an element to a machine word, and usually are not as scalable [11][12]. Gottlieb et al. described an array-based algorithm without these limitations, but their design is blocking [13]. Channels use conventional `CAS`, encapsulate user-defined data (of any size), are non-blocking, and scale well on GPUs.

9.2 Dynamic Aggregation for Data-parallel Hardware

GRAMPS, which inspired channels, maps flow graphs to graphics pipelines and provides packet queues to aggregate fine-grain work into data-parallel tasks [20]. Channels apply these concepts to more general computation. Our work gives a fresh perspective on how to implement aggregation queues and use them to realize higher-level languages on GPUs.

Dynamic micro-kernels allow programmers to regroup threads using the keyword `spawn` [32]. To support this semantic, a fully associative look-up table (LUT), indexed on the program counter of the branch destination, is proposed. While micro-kernels target mitigating branch divergence, they could be used for dynamic work aggregation. Compared to channels, one limitation is that the number of tasks is limited to the number of entries in the LUT.

Stream compaction uses global scan and scatter operations to regroup pixels by their consumption kernels [33]. Channels avoid regrouping by limiting each channel to one consumption function.

The Softshell GPU task runtime uses persistent GPU workgroups to schedule and aggregate work from a monolithic task queue [34]. Channels instantiate a separate queue for each consumption function and leverage the GPU's control processor to manage those queues.

9.3 GPU Tasks and Synchronization

Aila and Laine proposed a scheme that they call persistent threads, which bypasses the GPU scheduler and places the scheduling burden directly on the programmer [35]. Exactly enough threads are launched to fill the machine and poll a global work queue. In contrast, channels fill the machine in a data-flow manner and only launch consumers that will dequeue the same work, which encourages higher SIMT utilization.

Tzeng et al. also explored task queues within the confines of today's GPUs [36]. Their approach was to operate on a queue at wavefront granularity. They allocated a queue per SIMD unit and achieved load-balance through work stealing/sharing. Channels use dynamic aggregation to provide a more conventional task abstraction.

Heterogeneous System Architecture (HSA) supports dependencies among kernels [5]. Similarly, dynamic parallelism in CUDA enables coarse-grain work coordination [6]. These approaches require programmers to reason about parallelism at a coarse granularity. We found that specifying dependencies at a coarse granularity, while scheduling work at a fine granularity, worked well for Cilk.

Fung et al. proposed that a wavefront's state be checkpointed to global memory for the purposes of recovering from a failed transaction [37]. We propose a wavefront yield instruction to facilitate Cilk on GPUs. While similar, we go a step further by allowing the wavefront to relinquish its execution resources. In contrast, CUDA and HSA only support context switches at kernel granularity.

10. Conclusion

Channels aggregate fine-grain work into coarser-grain tasks that run efficiently on GPUs. This section summarizes our work on channels, discusses its implications, and anticipates future work.

We proposed the first channel implementation. While our design scales to eight compute units, there are several improvements that future work should consider. Our implementation is a flat queue, but a hierarchical design may scale even better. We also used a read-for-ownership coherence protocol in our evaluation, but future work should quantify the effects of write-combining caches. Finally, future designs should optimize the layout of CEs in memory for SIMT hardware.

We described a set of transformations to map Cilk to a channel-flow graph. Future work should investigate mapping other high-level languages to GPUs through channels. Our implementation of Cilk on top of channels hard-codes both Cilk's dependencies and the subset of channels from which to schedule in the aggregator's firmware. Future work should explore general abstractions for managing the channels and their dependencies. For example, it may be possible to apply the concept of guarded actions to channels [38].

We used the GPU's control processor, which we called the aggregator, to manage channels and restart yielded wavefronts. We found that its architecture had a crucial impact on the performance of channels. While the out-of-order design that we used worked well for our workloads, a more efficient design might achieve similar results. Future work should explore control processor architectures that enable other novel GPGPU programming models and abstractions.

Acknowledgements

We thank Benedict Gaster and Michael Mantor for their critical input early on, Rex McCrary for explaining the control processor, and Yasuko Eckert for helping to define the aggregator configurations. We also thank Michael Boyer, Shuai Che, Mark Hill, Lee Howes, Mark Nutter, Jason Power, David Schlosser, Michael Schulte, and the anonymous reviewers for improving the presentation of the final paper. This work was performed while Marc Orr was a graduate student intern at AMD Research. Prof. Wood serves as a consultant for and has a significant financial interest in AMD.

References

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Proc. IEEE*, vol. 86, no. 1, pp. 82–85, Jan. 1998.

[2] S. R. Gutta, D. Foley, A. Naini, R. Wasmuth, and D. Cherepacha, "A Low-Power Integrated x86-64 and Graphics Processor for Mobile Computing Devices," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2011, pp. 270–272.

[3] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, "A Fully Integrated Multi-CPU, GPU and Memory Controller 32nm Processor," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2011, pp. 264–266.

[4] "Bringing High-End Graphics to Handheld Devices," NVIDIA, 2011.

[5] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," AMD, Aug. 2012.

[6] "CUDA C Programming Guide." [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[7] "OpenCL 2.0 Reference Pages." [Online]. Available: http://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/.

[8] B. R. Gaster and L. Howes, "Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck?" *Computer*, vol. 45, no. 8, pp. 42–52, Aug. 2012.

[9] "Intel Threading Building Blocks." [Online]. Available: http://www.threadingbuildingblocks.org/.

[10] S. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," in *Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[11] J. Valois, "Implementing Lock-Free Queues," in *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, 1994, pp. 64–69.

[12] C. Gong and J. M. Wing, "A Library of Concurrent Objects and Their Proofs of Correctness," Carnegie Mellon University, Technical Report, 1990.

[13] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM Trans Program Lang Syst*, vol. 5, no. 2, pp. 164–189, Apr. 1983.

[14] M. M. Michael and M. L. Scott, "Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared Memory Multiprocessors," *J Parallel Distrib Comput*, vol. 51, no. 1, pp. 1–26, May 1998.

[15] E. Ladan-mozes and N. Shavit, "An Optimistic Approach to Lock-Free FIFO queues," in *Proceedings of the 18th International Symposium on Distributed Computing*, 2004, pp. 117–131.

[16] "HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG)," HSA Foundation, Spring 2013.

[17] M. Harris, "Optimizing Parallel Reduction in CUDA," NVIDIA. [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[18] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[19] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proceedings of the 11th International Conference on Compiler Construction*, London, UK, 2002, pp. 179–196.

[20] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "GRAMPS: A Programming Model for Graphics Pipelines," *ACM Trans Graph*, vol. 28, no. 1, pp. 4:1–4:11, Feb. 2009.

[21] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, New York, N.Y., USA, 1998, pp. 212–223.

[22] G. Diamos and S. Yalamanchili, "Speculative Execution on Multi-GPU Systems," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[23] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism," in *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*, 2009, pp. 1–12.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Comput Arch. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[25] P. Conway and B. Hughes, "The AMD Opteron Northbridge Architecture," *IEEE Micro*, vol. 27, no. 2, pp. 10–21, Mar. 2007.

[26] B. A. Hechtman and D. J. Sorin, "Exploring Memory Consistency for Massively-Threaded Throughput-Oriented Processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, New York, N.Y., USA, 2013, pp. 201–212.

[27] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "QuickRelease: A Throughput-oriented Approach to Release Consistency on GPUs," presented at the 20th IEEE International Symposium On High Performance Computer Architecture (HPCA-2014).

[28] V. Strassen, "The Asymptotic Spectrum of Tensors and the Exponent of Matrix Multiplication," in *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, Washington, D.C., USA, 1986, pp. 49–54.

[29] AMD Corporation, "AMD Accelerated Parallel Processing SDK." [Online]. Available: http://developer.amd.com/tools-and-sdks/.

[30] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2009. ISPASS 2009*, 2009, pp. 163–174.

[31] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Thread Scheduling for Massively Multithreaded Processors," *IEEE Micro*, vol. 33, no. 3, pp. 78–85, May 2013.

[32] M. Steffen and J. Zambreno, "Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, D.C., USA, 2010, pp. 237–248.

[33] J. Hoberock, V. Lu, Y. Jia, and J. C. Hart, "Stream Compaction for Deferred Shading," in *Proceedings of the Conference on High Performance Graphics 2009*, New York, N.Y., USA, 2009, pp. 173–180.

[34] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg, "Softshell: Dynamic Scheduling on GPUs," *ACM Trans Graph*, vol. 31, no. 6, pp. 161:1–161:11, Nov. 2012.

[35] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of the Conference on High Performance Graphics 2009*, New York, N.Y., USA, 2009, pp. 145–149.

[36] S. Tzeng, A. Patney, and J. D. Owens, "Task Management for Irregular-Parallel Workloads on the GPU," in *Proceedings of the Conference on High Performance Graphics*, Aire-la-Ville, Switzerland, 2010, pp. 29–37.

[37] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, N.Y., USA, 2011, pp. 296–307.

[38] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, New York, N.Y., USA, 2013, pp. 142–153.